CS335A: Milestone 2

Sawan H N (210952) Shrilakshmi S K (211012) Yashas D (211199)

April 1, 2024

1 Required Environment

For the environment setup, it is essential to have Flex and Bison installed. On Linux systems, these tools can be installed using the commands `sudo apt install flex` for Flex and `sudo apt install bison` for Bison, respectively. Additionally, the system must have the `g++` compiler available for use.

2 Instructions to run the code:

Go to directory `cs335-project-39` and run the following commands in terminal:

```
cd milestone2
./make.sh
./run.sh -i <input.py>
```

All the `.csv` files of symbol tables and 3AC code `3AC.txt` will be outputted in `milestone2` directory. Note that `make` might fail sometimes on Windows. Please retry if it does. On Linux, if any file denies permission, please execute `sudo chmod +x <filename>` for that file.

3 Options

How to use: `./run.sh` [options]
Options:

```
-i Specify input (.py) file
-v Enable verbose mode
-h Show help
```

4 Changes to Grammar

We have introduced some constraints on official Python 3.8 grammar. LHS of the expression statements cannot be constants. Further, we have divided the expression statements into initialization expression statements and non-initialization expression statements.
Following are the valid LHS for initialization expression statements:

- `NAME`
- `NAME DOT NAME` (where the former NAME can only be "self")

Following are the valid LHS for non-initialization expression statements:

- `NAME`
- `NAME DOT NAME`
- `NAME [test]`
- `NAME DOT NAME [test]`

Further we have following expression statements with no RHS.

- `test`
- `NAME (arglist)`
- `NAME DOT NAME (arglist)`

We have handled all the above cases.

5 Symbol Table

We have created different symbol tables for different scopes. Class, Function and Global are the scopes. We do not have different scopes for control statement blocks, as it was unnecessary. We store token, type, offset, scope, initialization status, number of parameters and parameter number in symbol table.

If a class is child of another class, we copy all the symbol table entries of parent class and proceed with populating the table further.

6 Supported Features

- Class inheritance: Child classes can access parent class's attributes and methods. While accessing methods, type of `self` argument is checked.
- Method call by both objects and classes

```
1  class Student:
2      def __init__(self, name : str, age : int):
3          self.name : str = name
4          self.age : int = age
5      def get_grade(self, num : int) -> int:
6          grade : int = self.age - num
7          return grade
8
9  student1 : Student = Student("Shril", 20)
10 grade_obj : int = student1.get_grade(13)  # method
   ↪ call by object without 'self' argument
11 grade_class : int = Student.get_grade(student1, 13)
   ↪ # method call by class with 'self' argument as
   ↪  object
```

- Implicit typecasting of `int <---> bool` and `int <---> float`. This typecasting is supported for expression evaluations, assignments, parameters type matching and return value from functions. For expression evaluations, if types of two operands differ, we have always typecasted `int` to `float` and `bool` to `int`. During assignment, they might

get typecasted back depending upon the type hint given for LHS. We have also supported typecasting of child class type into parent class type.

- We have implemented the `range()` function in our 3AC code.
- We have supported type checking in all possible scenarios: All the assignment expressions stated in previous section, return values, parameter types including `self` parameter while also supporting child typecast to parent.
- Global variables have been supported. Our version of global variable means that they cannot be declared again in any scope.
- `break` and `continue` statements are supported in for and while loops.

7 3AC Code Generation

7.1 Expression Statements

We followed an analogy similar to AST construction in creating 3AC code for expression statements. We created a new temporary variable if there was an operation in the production of the non-terminal, else we passed on the temporary variable to higher non-terminals in the parse tree.

```
1 def main():
2     x : int = 2 * 3 + 4 / 5 ** 9
```

```
1  main:
2      beginfunc
3      push %rbp
4      %rbp = %rsp
5      t0 = 2
6      t1 = 3
7      t2 = t0 * t1
8      t3 = 4
9      t4 = 5
10     t5 = 9
11     t6 = t4 ** t5
12     t7 = t3 / t6
```

```
    t8 = t2 + t7
    x = t8
    sub %rsp 4
    -4(%rbp) = x
    add %rsp 4
    pop %rbp
    ret
    endfunc
```

7.2 Control Flow Statements

For "if-elif-else" statements, we have used a stack to store the exit labels of elif statements. For "for statements", we have internally implemented the `range()` function.

7.2.1 If Statement

```
def main():
    a : int = 2
    if a > 3:
        a = 1
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = 2
    a = t0
    sub %rsp 4
    -4(%rbp) = a
    t1 = 3
    t2 = a > t1
    Ifz t2 Goto L0
    t3 = 1
    a = t3
    -4(%rbp) = a
    Goto L1
L0:
```

```
L1:
     add %rsp 4
     pop %rbp
     ret
     endfunc
```

7.2.2 While Statement

```
def main():
     a : int = 2
     while a < 1:
          a -= 1
```

```
main:
     beginfunc
     push %rbp
     %rbp = %rsp
     t0 = 2
     a = t0
     sub %rsp 4
     -4(%rbp) = a
L0:
     t1 = 1
     t2 = a < t1
     Ifz t2 Goto L1
     t3 = 1
     t4 = a - t3
     a = t4
     -4(%rbp) = a
     Goto L0
L1:
     add %rsp 4
     pop %rbp
     ret
     endfunc
```

7.2.3 For Statements

```
def main():
    a : int = 3
    i : int
    b : int = 0
    for i in range(a):
        b = i
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = 3
    a = t0
    sub %rsp 4
    -4(%rbp) = a
    t1 = 0
    b = t1
    sub %rsp 4
    -12(%rbp) = b
    t2 = 0
    i = t2
    Goto L1
L0:
    t5 = i + 1
    i = t5
L1:
    t6 = i < a
    Ifz t6 Goto L2
    b = i
    -12(%rbp) = b
    Goto L0
L2:
    add %rsp 12
    pop %rbp
    ret
    endfunc
```

7.3 Lists

Since list size cannot be pre-determined, especially when it is a parameter in a function, we have pointers for lists. List size, then becomes 8 bytes - the size of a pointer. We first call `allocmem` and allocate the memory for list and initialize the elements of the list one by one. As lists are mutable, this approach is useful to modify individual elements of list.

```
def main():
    a : int = 3
    i : int
    b : int = 0
    for i in range(a):
        b = i
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = 2
    t1 = 3
    t2 = 5
    t3 = 6
    t4 = 16
    pushparam t4
    push ret_addr
    sub %rsp 4
    call allocmem 1
    add %rsp 4
    t5 = %rax
    *(t5 + 0) = t0
    *(t5 + 4) = t1
    *(t5 + 8) = t2
    *(t5 + 12) = t3
    a = t5
    sub %rsp 8
    -8(%rbp) = a
    t6 = 0
    t7 = t6 * 4
    t8 = 1
    *(a + t7) = t8
```

```
27      add %rsp 8
28      pop %rbp
29      ret
30      endfunc
```

7.4 Objects

Similar to list, to construct objects, we allocate the size of object in heap using allocmem, when constructor function is called. We then assign the values of object attributes based on offsets stored in symbol table.

```
1  class Student:
2      def __init__(self, name : str, age : int):
3          self.name : str = name
4          self.age : int = age
5
6  def main():
7      s : Student = Student("Shril", 19)
8      s.age = 20
```

```
1  Student.__init__:
2      beginfunc
3      push %rbp
4      %rbp = %rsp
5      self = 16(%rbp)
6      sub %rsp 8
7      -8(%rbp) = self
8      name = 24(%rbp)
9      sub %rsp 8
10     -16(%rbp) = name
11     age = 32(%rbp)
12     sub %rsp 4
13     -20(%rbp) = age
14     *(self + 0) = name
15     *(self + 8) = age
16     add %rsp 20
17     pop %rbp
18     ret
19     endfunc
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = "Shril"
    t1 = 19
    t2 = 12
    pushparam t2
    sub %rsp 4
    call allocmem 1
    add %rsp 4
    t3 = %rax
    pushparam t1
    pushparam t0
    pushparam t3
    push ret_addr
    sub %rsp 20
    call Student.__init__ 3
    add %rsp 20
    s = t3
    sub %rsp 8
    -8(%rbp) = s
    t4 = 20
    *(s + 8) = t4
    add %rsp 8
    pop %rbp
    ret
    endfunc
```

7.5 Typecasting

We have used a method `cast_to_type` to typecast in 3AC code.

```
def add(a : int, b : float) -> float:
    c : int = a + b
    return c

def main():
    res : int = add(2.3, 3)
```

```
add:
    .........
    -8(%rbp) = b
    t0 = cast_to_float a
    t1 = t0 + b
    t2 = cast_to_int t1
    c = t2
    sub %rsp 4
    -12(%rbp) = c
    t3 = cast_to_float c
    ..........
main:
    ..........
    t4 = 2.3
    t5 = 3
    t6 = cast_to_int t4
    t7 = cast_to_float t5
    pushparam t7
    pushparam t6
    .........
```

8 Runtime Support

8.1 After function is called

When a function is called, we push `%rbp` on the stack. Then we are accessing the parameters of callee function (beginning from `16(%rbp)`) from the stack and storing them as local variables again. We are also pushing other local variables to the stack. We modify `%rsp` according to their widths. Whenever we modify these local variables, we are also modifying the value in the stack. For pointers such as objects and lists, this is not necessary as their value modifications gets reflected in stack too when we change them in the function. To return, we are storing the value of return value in `%rax` register. Then we pop all the local variables, we adjust `%rsp` accordingly. We then pop old `%rbp` and change `%rbp`. We lastly use instruction `ret` which pops the return address from the stack and `%rip` continues its execution from that address. **All of these register manipulations are reflected in our 3AC code.**

```
add:
    beginfunc
    push %rbp
    %rbp = %rsp
    a = 16(%rbp)
    sub %rsp 4
    -4(%rbp) = a
    b = 20(%rbp)
    sub %rsp 4
    -8(%rbp) = b
    ...........
```

8.2 Calling a function

To call a function, we first push the parameters of callee function on the stack using `pushparam`. We modify `%rsp` according to their widths. We then push return address. We then call the function using `call func_name num_params` line in 3AC code. **All of these register manipulations are reflected in our 3AC code.**

```
main:
    ...........
    t4 = 2
    t5 = 3.2
    pushparam t5
    pushparam t4
    push ret_addr
    sub %rsp 8
    call add 2
    add %rsp 8
    ..........
```

8.3 Returning from a function

If the return value exists, then it will be stored in `%rax` register. We assign this value to a temporary variable. Then we assign this temporary variable to the return variable. **All of these register manipulations are reflected in our 3AC code.**

```
main:
     ...........
     t6 = %rax
     res = t6
     ..........
```

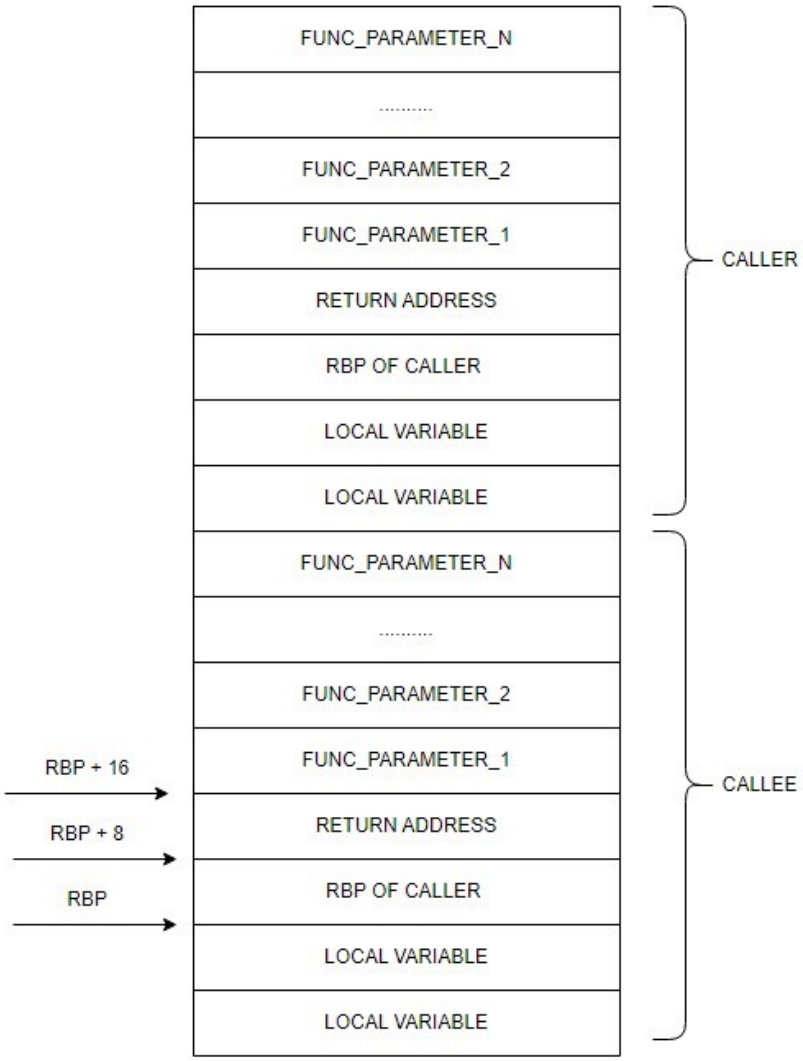

Figure 1: Stack for Runtime Support

9 Error Handling

This is the non-exhaustive list of errors reported with **separate** error messages. Error messages are reported at **160+** places in the parser code with line numbers.

- Redeclaration of any symbol table entry
- Redeclaration of global variable in any scope
- Global variable cannot be declared (`x : int = 5`) before global declaration (`global x`)
- Return type error based on the function's return type hint and function's return value. We have also handled the case of None return type
- Type hints for parameters not given
- Invalid assignments for LHS and RHS types
- `self` cannot be used as variable name. It is reserved for instances of classes.
- Number of arguments passed in functions and methods (including print(), len() and range())
- Types of arguments passed in functions and methods (including print(), len() and range()) including support for typecasting
- Undeclared function/method error
- Undeclared variable/attribute error
- **Uninitialised variable/attribute error** - If a variable/attribute is used in RHS without initialization of some value beforehand, error is reported
- Constructor call parameter errors
- Undeclared object error
- Undeclared class error

- Invalid operation error based on the operator and types of operands
- Attribute reinitialization error
- Initialization of self attribute in non-constructor function error
- List index type must be int
- Index error, for indexing non-list types
- Index error, for indexing lists with higher dimensions
- Iterating variable in for loop must be int
- Base / parent class does not exist error

10 Assumptions

- `__init__()` function / constructor does not have return type mentioned
- `self` does not have type mentioned in parameters
- No stray code in classes
- All the statements are enclosed in functions except for

```
1 if __name__ == "__main__":
2     main()
```

- Parent class must be defined before child class, else an error will be thrown
- Functions are defined before calling in any another function

11 Contribution

- Sawan - 33.3%
- Shrilakshmi - 33.3%
- Yashas - 33.3%